“Progress Report on Parallel Xpdc1”

Emi Kawamura

In electrostatic 1d3v PIC simulations of RF discharges, the field solve is typically less than 1 percent of the work load. Thus, we can obtain significant gains by just paralleling the particle processing (e.g., pushing/accumulating) without paralleling the field solve. I applied this simple parallelization scheme to xpdc1. For a fixed number of grid points, the speedup for this parallel particle processing became more linear with increasing particle number. This paralleling method is also described in UCB/ERL report M99/58.

1 Design of Parallel PIC-MCC

In our scheme, the particles are randomly and equally divided among the CPUs regardless of their positions within the grid. Each CPU sees the entire spatial grid but only a random portion of the particles. Since the field solve is a small fraction of the simulation time, the total electron and ion densities are broadcast among all the processors, and the field solve is done redundantly by each processor. The steps to the parallel particle processing scheme are as follows:

1. Each CPU advances its allotment of particles and linearly weights its particles to the grid.

2. The contributions of all the CPUs are summed to find the total densities of each species on the grid.

3. The CPUs communicate with each other to determine the total density for each species.(Recall that each CPU sees only a portion of the particles so that the total density for a species is the sum of the densities seen by the individual CPUs.) Then, each CPU can conduct its own field solve to advance its particles.

Note that this parallel particle processing method yields significant gains only if the field solve is a small fraction of the total work load.

As described above, the physical region for simulation is the same for each processor; (i.e., each processor sees the entire spatial grid). Also, all the particles are randomly and equally divided among the processors at the outset of the simulation. This implies that we get an automatic static load balancing: the amount of work done by each CPU is roughly the same even after many iterations.

This contrasts with the Eulerian decomposition scheme in which the physical region of simulation is divided among the processors. Any particles in a physical region are handled by the processor corresponding to that region. In such a scheme, a dynamic load balancing method must also be developed for optimal effect since some spatial partitions may accumulate more particles than others.

It is often desirable to store the state of a system in a dump file, so that a simulation can be restarted from the dump point rather than the starting point. To dump the state of the system in our parallel run, each CPU sends the positions and velocities of its particles to a root CPU which gathers the information and writes it to a file. When restoring, the root CPU reads the dump file and distributes the particles randomly among the CPUs.

2 MPI Libraries

Our parallel particle processing codes use the MPICH implementation of the MPI (Message Passing Interface) library. MPICH is developed at Argonne National Laboratory and Mississipi State University. (MPICH can be downloaded from the website http://www-unix.mcs.anl.gov/mpi/mpich).

The MPI Library contains a suite of useful functions for writing parallel codes in both C and Fortran. The following is a brief description of the MPI routines used in the parallel particle processing code. The routines use either a point to point communication procedure or a tree scheme. That is, a source CPU sends data directly to each of its target CPUs

(point to point scheme), or all the CPUs collectively participate in distributing the data (tree scheme). These schemes are displayed in Fig. 1. The text by Peter Pacheco, "Parallel Programing with MPI", is an excellent introduction to parallel programming with MPI and contains many useful references.

Note that for a point to point scheme the communication time increases as $N_{proc} - 1$, where N_{proc} is the number of processors, while, for a tree scheme, the time increases as $\log_2 N_{proc}$.

- **MPI_Bcast**: Broadcasts data collectively from a source CPU to many target CPUs (tree scheme).
- **MPI_Reduce**: Sums the data of CPUs together and sends the result to root (tree scheme).
- **MPI_Allreduce**: Sums the data of CPUs together and sends the result to *all* the CPUs (tree scheme).
- **MPI_Gatherv**: Gathers variable size arrays from different CPUs and concatenates them together (tree scheme).
- **MPI_Send & MPI_Receive**: Send and receive data from a source CPU to a target CPU (point to point scheme).

The MPI libraries can be configured for a particular computing environment. For example, the MPICH distribution of MPI comes with libraries configured for shared memory systems such as symmetric multiprocessors (SMPs). Also members of the UC Berkeley Network of Workstations (NOW) project, headed by Prof. David Culler, have written MPI libraries optimized for their distributed network of Sun Ultra 170 workstations[**?**]. Using the libraries especially configured for the computing environment significantly reduces the communication time among the CPUs.

In our simulation, MPI_Send and MPI_Receive are used only in the initial stage of a simulation when particles are restored from a dump file and divided up among the CPUs.

MPI_Gatherv is only used when gathering data to display diagnostics or to store in a dump file.

MPI_Allreduce is used to sum the electron and ion density data of all the CPUs and send the result to *all* the CPUs. Then, each CPU individually calculates the field solve to advance its particles. Note that except for the initial stage of the simulation, we primarily use a tree communication scheme.

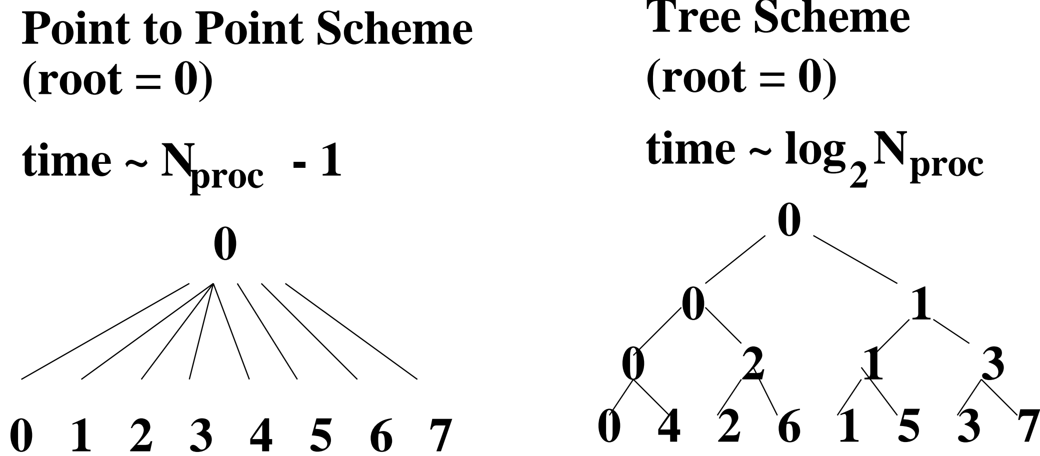

Figure 1: A point to point scheme communication model compared to a tree scheme communication model. From Pacheco's "Parallel Programming with MPI".

3 Test of Parallel Xpdc1

I encoded the parallel particle processing scheme to xpdc1 and tested it on a 2 CPU Intel Pentium II symmetric multiprocessor (SMP) machine using the shared memory (ch_shmem) MPI library.

```
*******************************************************************
When running the parallel code for 100 timesteps without X, I get:


alfven14%shmpirun -np 2 xpdc1.par.sh -i radial.inp -s 100 -nox
Using Parallel Version of PDC1
```

```
PDC1 - Cylindrical Bounded Electrostatic 1 Dimensional Code
version 3.0
(c) Copyright 1989-95 Regents of the University of California
Plasma Theory and Simulation Group
University of California - Berkeley
See README file for information about Licensing


Input file is 'radial.inp'
Running for 100 steps
Running without X
 n_pk = 1.000000e+17
np = 176348 maxnp = 250000 nc2p = 5.000000e+07
 n_pk = 1.000000e+17
np = 176348 maxnp = 250000 nc2p = 5.000000e+07
proc 0: Species 0: 87968 particles loaded
proc 0: Species 1: 87968 particles loaded
sp_n_tot[0][ng/2] = 1.70668
sp_n_tot[1][ng/2] = 1.70668
proc 0: sp_n[0][ng/2] = 0.853339
proc 0: sp_n[1][ng/2] = 0.853339
wall clock time = 33.078738
init time = 0.941821
Finalizing MPI
mem=0 pf=1874 swap=0 33.080u 0.100s 0:33.77 0+0io

*****************************************************************
When running the non-parallel code on two CPUs for 100 timesteps
```

```
without X, I get:

alfven17%xpdc1 -i radial.inp -s 100 -nox


PDC1 - Cylindrical Bounded Electrostatic 1 Dimensional Code
version 3.0
(c) Copyright 1989-95 Regents of the University of California
Plasma Theory and Simulation Group
University of California - Berkeley
See README file for information about Licensing


Input file is 'radial.inp'
Running for 100 steps
Running without X
 n_pk = 1.000000e+17
np = 176348 maxnp = 250000 nc2p = 5.000000e+07
 n_pk = 1.000000e+17
np = 176348 maxnp = 250000 nc2p = 5.000000e+07
Species 0: 175930 particles loaded
Species 1: 175930 particles loaded
mem=0 pf=220 swap=0 65.310u 0.030s 1:05.45 0+0io

****************************************************************
```

Thus, the gain in time is $65.13/33.08 = 1.97$ when using 2 CPUS.

3.1 Test of Parallel Dump and Restore

I've tested the parallel dump and restore, and it appears to work.

The following is a test conducted on a 4 CPU pentium pro SMP machine.

```
**********************************************************************

First I run on 4 CPUs for 100 timesteps and dump:

panoply [34] shmpirun -np 4 -machinefile machines.pan xpdc1.par -i
radial.inp -s 100 -nox -dp 100
Using Parallel Version of PDC1



PDC1 - Cylindrical Bounded Electrostatic 1 Dimensional Code
version 3.0
(c) Copyright 1989-95 Regents of the University of California
Plasma Theory and Simulation Group
University of California - Berkeley
See README file for information about Licensing



Input file is 'radial.inp'
Running for 100 steps
Running without X
Dumping every 100 steps
 n_pk = 1.000000e+17
np = 176348 maxnp = 250000 nc2p = 5.000000e+07
 n_pk = 1.000000e+17
np = 176348 maxnp = 250000 nc2p = 5.000000e+07
```

```
proc 0: Species 0: 43984 particles loaded
proc 0: Species 1: 43984 particles loaded
sp_n_tot[0][ng/2] = 1.70782
sp_n_tot[1][ng/2] = 1.70782
proc 0: sp_n[0][ng/2] = 0.426954
proc 0: sp_n[1][ng/2] = 0.426954
Start Dump
np_tot[0] = 175948
proc 0:  np[0] = 43987
np_tot[1] = 175952
proc 0:  np[1] = 43988
sp_n_tot[0][ng/2] = 1.58081
sp_n_tot[1][ng/2] = 1.70777
proc 0: sp_n[0][ng/2] = 0.395202
proc 0: sp_n[1][ng/2] = 0.426944
Dump done
wall clock time = 37.222675
init time = 1.002590
Finalizing MPI
```

****

Next I do a restore from a dump file and run for an additional 100 timesteps on 4 CPUs.

```
panoply [35] shmpirun -np 4 -machinefile machines.pan xpdc1.par -i
radial.inp -s 100 -nox -d radial.dmp
Using Parallel Version of PDC1
```

```
PDC1 - Cylindrical Bounded Electrostatic 1 Dimensional Code
version 3.0
(c) Copyright 1989-95 Regents of the University of California
Plasma Theory and Simulation Group
University of California - Berkeley
See README file for information about Licensing


Input file is 'radial.inp'
Running for 100 steps
Running without X
Dump file is 'radial.dmp'
Reading Restore:
time = 3e-10
nsp = 2
np_tot[0] = 175948
np_tot[1] = 175952
proc 1 to proc 3: isp = 0; np = 43987
proc 0: isp = 0; np = 43987
proc 1 to proc 3: isp = 1; np = 43988
proc 0: isp = 1; np = 43988
End Restore
sp_n_tot[0][ng/2] = 1.58081
sp_n_tot[1][ng/2] = 1.70778
proc 0: sp_n[0][ng/2] = 0.392032
proc 0: sp_n[1][ng/2] = 0.425263
wall clock time = 37.766584
init time = 2.802993
Finalizing MPI
```

Comparing the diagnostics information after the end of the dump with the diagnostics information at the end of the restore, we see a good match. The total densities (sp_n_tot) and the total number of particles (np_tot), the number of particles seen by processors 1-3, and the number of particles seen by the root CPU (proc. 0) all match up.

The only difference is the densities seen by the root CPU (proc 0: sp_n). This is because the particles are randomly redistributed among the CPUs in the restore phase. That means the root CPU does not see the same particles it did in the second run as it did in the first run. But all the particles are accounted for as indicated by the matching sp_n_tot and np_tot.

Also, when running radial.inp with 4 CPUs 100 timesteps without X, I got a Gain = 3.86 over running with 1 CPU. So I have almost linear gain with the 4 CPU and 2 CPU SMP machines.

The dump files are fully compatible between the non-parallel and parallel versions. This means that we can run in parallel without X until equilibrium and dump to a file. Then we can restart a non-parallel simulation with X from the dumpfile.

After I have fully tested this parallel xpdc1 code. I will incorporate the code into the CVS repository.